AALBORG UNIVERSITY
DENMARK

Programming Exercise: Socket Programming

In order to get some hands-on expertise in socket programming and multi-threading, you will be implementing a simple Internet Radio Station that streams (multicasts) songs.
This assignment has two parts: the server, which streams songs to a multicast group, and a pair of clients for connecting to the server and receiving the songs.
There are two kinds of data being sent between the server and the client: One is the **control data**: The client uses this data to learn about the stations and the server uses it to give the client song information. The other kind is the **song data**, which the server reads from song files and streams to the multicast group. You will be using TCP for the control data and UDP for the song data.
The client and server communicate control data by sending each other messages over the TCP connection.

You can work in groups on this exercise, and are welcome to use any language you like. Moreover, if you believe that a different interface / set of commands than the one described below, could be more meaningful, you are encouraged to go for your version.

Client to Server Commands

The client sends the server messages called ***commands***. There are two commands the client can send the server, in the following format.

Hello:
int commandType = 0;

AskSong:
int commandType = 1;
int stationNumber;

The *Hello* command is sent when the client connects to the server. The ***AskSong*** command is sent by the client to inquire which song is played now in a station. **stationNumber** identifies the station the client inquire about.

Se*rver to Client Replies*

There are three possible messages called replies the server may send to the client:

Welcome:
int replyType = 0;

int numStations;
int multicastGroup;
int portNumber;

***Announce**:*
int replyType = 1;
int songnameSize;
char songname[songnameSize]; ***InvalidCommand**:*
int replyType = 2;
int replyStringSize;
char replyString[replyStringSize];

A ***Welcome*** reply is sent in response to a ***Hello*** command. Stations are numbered sequentially from 0, so a **numStations** of 30 means 0 through 29 are valid. **multicastGroup** indicates the multicast IP address used for station 0. Station 1 will use the multicast group IP +1, etc. ***portNumber*** indicates the port to listen to. All stations are using the same port. A ***Hello*** command, followed by a ***Welcome*** reply, is called a *handshake*.

An ***Announce*** reply is sent after a client sends a ***AskSong*** command about a station ID. **songnameSize** represents the length, in bytes, of the filename, while **songname** contains the filename itself. So, to ***announce*** a song called *Gimme Shelter*, your server would send the **replyType** byte, followed by a byte whose value is 13, followed by the bytes whose values are the ASCII character values of *"Gimme Shelter"*.

Invalid Conditions

Since neither the client nor the server may assume that the program with which it is communicating is compliant with this specification, they must both be able to behave correctly when the protocol is used incorrectly.

On the server side, an ***InvalidCommand*** reply is sent in response to any invalid command. *replyString* should contain a brief error message explaining what went wrong. Give helpful strings stating the reason for failure. If a ***AskSong*** command was sent with 1729 as the *stationNumber*, a bad *replyString* is “Error, closing connection.”, while a good one is “Station 1729 does not exist”. To simplify the protocol, whenever the server receives an invalid command, it must reply with an ***InvalidCommand*** and then close the connection to the client that sent it.

Invalid commands happen in the following situations:
• AskSong
– The station given does not exist.
– The command was sent before a ***Hello*** command was sent. The client must send a ***Hello*** command before sending any other commands.
– If the command was sent before the server responded to a previous ***AskSong*** by sending an ***Announce*** reply, then your server MAY reply to this with an ***InvalidCommand***. This means that your client should be careful and wait for an ***Announce*** before sending another ***AskSong***, but your server can be lax about this.
• Hello
– More than one ***Hello*** command was sent. Only one should be sent, at the very beginning.
• An unknown command was sent (one whose *commandType* was not 0 or 1).

On the client side, invalid uses of the protocol MUST be handled simply by disconnecting. This happens in the following situations:
• ***Announce***
– The server sends an ***Announce*** before the client has sent a ***AskSong***.
• ***Welcome***
– The server sends a ***Welcome*** before the client has sent a ***Hello***.

– The server sends more than one ***Welcome*** at any point (not necessarily directly following the first ***Welcome***).
• ***InvalidCommand***
– The server sends an ***InvalidCommand***. This may indicate that the client itself is incorrect, or the server may have sent it out of error. In either case, the client MUST print the *replyString* and disconnect.
• An unknown response was sent (one whose *replyType* was not 0, 1, or 2).

Sometimes, a host you're connected to may misbehave in such a way that it simply doesn't send any data. In such cases, it's imperative that you are able to detect such errors and reclaim the resources consumed by that connection. In light of this, there are a few cases in which you will be required to time out a connection if data isn't received after a certain amount of time.

These timeouts should be treated as errors just like any other I/O error you might have, and handled accordingly. In particular, they must be taken to only affect the connection in question, and not unrelated connections (this is obviously more of a problem for the server than for the client).

The requirements related to timeouts are:
• A timeout may occur in any of the following circumstances:
– If a client connects to a server, and the server does not receive a ***Hello*** command within some preset amount of time, the server should time out that connection. If this happens, the timeout must not be less than 100 milliseconds.
– If a client connects to a server and sends a ***Hello*** command, and the server does not respond with a ***Welcome*** reply within some preset amount of time, the client should time out that connection. If this happens, the timeout must not be less than 100 milliseconds.
– If a client has completed a handshake with a server, and has sent a ***AskSong*** command, and the server does not respond with an ***AskSong*** reply within some preset amount of time, the client should time out that connection. If this happens, the timeout most not be less than 100 milliseconds.
• A timeout must not occur in any circumstance not listed above.

Note that while we specify precise times for these timeouts, we don't expect your program to behave with absolute precision. Processing delays and constraints of running in a multi-threaded environment, among other concerns, make such precision guarantees impossible. We simply expect that you make an effort to come reasonably close - don't be off by wide margins when you could make obvious improvements, but also don't bother trying to finely tune it.

You should write two separate clients.
(1) The UDP client handles song data.
(2) The TCP client handles the control data.

The control client must connect to the server and communicate with it according to the protocol. After the handshake, it must wait for input. If the user types in a number, the control must send a ***AskSong*** command with the user-provided station number, unless that station number is outside the range given by the server; you may choose how to handle this situation. If the client gets an invalid reply from the server (one whose ***replyType*** is not 0, 1, or 2), then it must close the connection and exit.

Regarding the server, to make things easy, each station will contain just one song. Station 0 plays the first file, Station 1 plays the second file, etc... Each station must loop its song indefinitely.
When the server starts, it must begin listening for connections. When a client connects, it must interact with it as specified by the Protocol. You want the server to stream music, not to send it as fast as possible. Assume that all mp3 files are 128 Kibps, meaning that the server should send data at a rate of 128Kibps (16 KiB/s).

Additionally:

• The server must support multiple clients simultaneously.
• The server must never crash, even when a misbehaving client connects to it. The connection to that client may be terminated, however.
• If multiple clients are connected to one station, they must all be listening to the same part of the song, even if they connected at different times.
• If no clients are connected, the current position in the songs MUST still progress, without sending any data. The radio doesn't stop when no one is listening.
• The server must not simply read the entire song file into memory at once. It MAY read the entire file in for some sizes, but there must be a size beyond which it will be read in chunks.
• **Make sure** you close the socket whenever a client connection is closed, or when the program terminates. You must implement it both on the server and client.

Tipp: If you have problems using mp3 files, you can also simply stream text files instead.

You do not have to hand in your code and there is no deadline. However, in the oral exam, we will make a lottery and one question will ask you to report on your insights from this programming exercise.